QuickChecking static analysis properties

Midtgaard, Jan; Møller, Anders

Published in:
Software Testing, Verification and Reliability

Link to article, DOI:
10.1002/stvr.1640

Publication date:
2017

Document Version
Early version, also known as pre-print

SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 0000; **00**:1–29
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

QuickChecking Static Analysis Properties

Jan Midtgaard[1]*and Anders Møller[2]

[1]*DTU Compute, Technical University of Denmark*
[2]*Department of Computer Science, Aarhus University*

SUMMARY

A static analysis can check programs for potential errors. A natural question that arises is therefore: who checks the checker? Researchers have given this question varying attention, ranging from basic testing techniques, informal monotonicity arguments, thorough pen-and-paper soundness proofs, to verified fixed point checking. In this paper we demonstrate how quickchecking can be useful to test a range of static analysis properties with limited effort. We show how to check a range of algebraic lattice properties, to help ensure that an implementation follows the formal specification of a lattice. Moreover, we offer a number of generic, type-safe combinators to check transfer functions and operators on lattices, to help ensure that these are, e.g., monotone, strict, or invariant. We substantiate our claims by quickchecking a type analysis for the Lua programming language. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Static Program Analysis; QuickChecking; Monotonicity; Domain-Specific Languages

1. INTRODUCTION

Fundamentally, most static analyses boil down to monotone functions operating over lattices [1]. To gain confidence in a static analysis implementation, one would thus hope that the code implements, at least, (1) the formal specification of being a lattice, and (2) functions that are in fact monotone. For example, consider a simple two-element lattice expressed as an OCaml module:

```
module L = struct
  let name = "example␣lattice"
  type elem = Top | Bot
  let leq a b = match a,b with
    | Bot, _ -> true
    | _, Top -> true
    | Top, Bot -> false
  let join e e' = if e = Bot then e' else Top
  let meet e e' = if e = Bot then Bot else e'
  (* ... *)
  let to_string e = if e = Bot then "Bot" else "Top"
end
```

*Correspondence to: J. Midtgaard, DTU Compute, Richard Petersens Plads bldg. 324, DK-2800 Kgs. Lyngby, Denmark. E-mail: mail@janmidtgaard.dk

Contract/grant sponsor: This work was supported by the Danish Research Council for Technology and Production (FTP) and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544).

Here, the module `L` defines a lattice with an algebraic data type `elem` for describing the two kinds of elements: `Top` and `Bot`. In addition, `L` defines a number of operations including `leq` (element ordering), `join` (least upper bound), and `meet` (greatest lower bound). In particular, `leq` utilizes pattern matching (dispatch over algebraic data types, with underscore `_` as catch-all) over pairs to cover all cases. How can we be sure that this module correctly implements a lattice? This requires the operations to satisfy various algebraic properties, for example, that `leq` is reflexive, transitive, and anti-symmetric.

We present a framework to effectively test implementations of lattices and their operations, which are an essential part of static analysis tools. Provided we extend the lattice module with a generator of arbitrary elements, e.g., by choosing arbitrarily among the list of values `[Bot; Top]`:

```
let arb_elem = Arbitrary.among [Bot; Top]
```

our framework provides a range of property tests to boost confidence in the implementation. For one we can pass the module `L` to the generic functor `GenericTests` to get back a new module, containing a runnable test suite (here shown with the user's input in bold font):

```
# let module LTests = GenericTests(L) in
  run_tests LTests.suite;;
  check 19 properties...
testing property leq reflexive in example lattice...
  [✓] passed 1000 tests (0 preconditions failed)
testing property leq transitive in example lattice...
  [✓] passed 1000 tests (0 preconditions failed)
testing property leq anti symmetric in example lattice...
  [✓] passed 1000 tests (0 preconditions failed)
... (32 additional lines cut)...
tests run in 0.02s
[✓] Success! (passed 19 tests)
```

Furthermore, to ensure that a static analysis implementation is guaranteed to reach a fixed point when analyzing a given program, the involved lattice operations should be monotone. For example, consider the following non-monotone operation `flip`, which maps bottom to top and top to bottom:

```
let flip e = if e = L.Bot then L.Top else L.Bot
```

In this paper we provide a type-safe, embedded domain-specific language (EDSL) to check and catch such operator errors, by expressing property signatures in a syntax that resembles the standard mathematical syntax, $\texttt{flip} : L \xrightarrow{\sqsubseteq} L$. We get runnable tests of such generic operator properties by providing only a signature and a description pair (consisting of a string and the function value of the operator):

```
# let flip_desc = ("flip",flip) in
  run (testsig (module L) -<-> (module L) =: flip_desc);;
  testing property 'flip monotone in argument 1'...
  [✗] 270 failures over 1000 (print at most 1):
  (Bot, Top)
```

(The operators `-<->` that expresses monotonicity and `=:` that builds the test are explained in detail in Section 4.) We only need two inputs (top and bottom) to achieve full coverage of `flip`'s implementation. However, testing each individual call to `flip` in isolation is not enough to uncover such an error and falsify monotonicity. Instead a test would need to make two calls on related inputs and compare their results. Undoubtedly, this is a simplistic example, but the need to test implementations still stands. Today's static analyses can establish interesting properties about programs in higher-order, dynamically typed languages, for example JavaScript. The intricate semantics of such languages induces complexity in the underlying lattices and in the operations over these. In this paper we demonstrate how *quickchecking* [2] can be used as an effective lightweight methodology to test a range of algebraic properties in static analyses. In situations where lattices and transfer functions are subject to change, for example, in the design phase or in the revision of an analysis, the approach can become a valuable tool. Our approach can also act as a supplement to pen-and-paper proofs or mechanized reasoning within a proof assistant. Towards this goal, this paper makes the following contributions:

DOI: 10.1002/stvr

- We explore how the ideas in quickchecking (briefly summarized in Section 2) can be applied to static analysis.
- We demonstrate how to lift generators of simple lattices to generators of composite lattices and how to use the generators to check a number of fundamental lattice properties expressed as a reusable lattice test suite (Section 3).
- We formulate a type-safe EDSL of property signatures for testing operations over lattices for a number of desirable properties (Section 4).
- We present a case study of quickchecking a nontrivial static type analysis for Lua, where the tests supplement a basic test suite of hand-written programs to collectively achieve nearly full coverage (Section 5).

Compared to the conference version [3], this paper

- describes the Lua type analysis in more detail,
- discusses a revision of the analysis implementation following a generator revision,
- illustrates how the approach also works for a range of numeric abstract domains,
- elaborates on the implementation and the signatures of the type-safe EDSL (including an extension for testing distributivity of operations),
- revises the experiment, which overall results in even better coverage, and
- contains an extended discussion of related work.

2. BACKGROUND

This section provides relevant background information on QuickCheck, Lua, and static type analysis.

2.1. A QuickCheck summary

Quickchecking [2] is a popular methodology within the functional programming community for performing *property-based testing*. The approach is based on the fundamental idea of *generating* tests rather than *hand-writing* them. The generated tests all share a common format described by a mathematical property. The property acts as a specification and is supposed to hold regardless of the input. For example, in the case of the `flip` operation, the property $\forall$`a,b. a` $\sqsubseteq$ `b` $\Rightarrow$ `flip a` $\sqsubseteq$ `flip b` captures monotonicity: for all pairs of ordered inputs, `flip` should preserve this ordering among its outputs. As such, a quickcheck test involves two components: (1) a *generator* for producing arbitrary input, and (2) *properties* that should be tested on the arbitrary input. Two domain-specific languages (DSLs) are used for this purpose, one for each component. The original QuickCheck technique was based on DSLs embedded into Haskell using a Haskell library [2]. Since then, the approach has been ported to numerous other programming languages, both statically and dynamically typed. The approach has also been extended beyond functional programming to test imperative (stateful) code [4], and has had a number of successes, e.g., using the commercial Erlang QuickCheck port to test and find numerous issues in automotive software [5]. For the remainder of this paper, we use the `qcheck` implementation of QuickCheck in OCaml. However, we stress that the approach is not specific to OCaml.

Suppose we wish to test the (incorrect) property from the introduction, that `flip` is a monotone function, hence satisfying the property $\forall$`a,b. a` $\sqsubseteq$ `b` $\Rightarrow$ `flip a` $\sqsubseteq$ `flip b`. By translating the universally quantified variables `a` and `b` into function parameters, this property can be expressed as an anonymous OCaml function with Boolean result type:

```
fun (a,b) -> Prop.assume (L.leq a b); L.leq (flip a) (flip b)
```

The implication of our specification is modeled using the operator `Prop.assume` from `qcheck`'s property DSL, which will test its precondition and (i) continue if it is **`true`**, or (ii) accept the test and bail early if it is **`false`** (while keeping track of the number of failed preconditions). This faithfully models logic implication: **`false`** implies anything. In general, properties concerning a value of type `'a` become predicates of type `'a -> bool`.

DOI: 10.1002/stvr

To test the above property on arbitrary pairs, we need to generate some input. Recall `Arbitrary.among : 'a list -> 'a Arbitrary.t` from the introduction: it is an example of a combinator from qcheck's generator DSL that will supply arbitrary elements selected from its argument list. We can lift this generator to a generator of pairs, using another built-in combinator, `Arbitrary.pair` (from here on we will abbreviate qcheck's `Arbitrary` module to `Arb`):

```
let arb_pair = Arb.pair L.arb_elem L.arb_elem
```

We are now in position to write a test with `mk_test` and subsequently run it, which exposes the error:

```
# let mon_test =
    mk_test arb_pair (fun (a,b) -> Prop.assume (L.leq a b); L.leq (flip a) (flip b));;
  val mon_test : QCheck.test = <abstr>
# run mon_test;;
testing property <anon prop>...
  [✗] 27 failures over 100
```

The error message neither names the failed property nor provides a counterexample. To do so, `mk_test` accepts a range of optional, named arguments (these are prefixed with tilde in OCaml). An example:

```
mk_test ~n:1000 ~pp:pp_pair ~name:"flip␣monotone"
  arb_pair (fun (a,b) -> Prop.assume (L.leq a b); L.leq (flip a) (flip b))
```

This will instead run the test on 1000 arbitrary pairs, it will identify the particular failed property by the supplied string `"flip␣monotone"`, and it will pretty-print up to ten counterexamples, using a supplied pretty-printer `pp_pair` (defined as a combination of its component's pretty-printers, `let pp_pair = PP.pair L.to_string L.to_string`).

2.2. *The Lua programming language*

Lua is a dynamically typed programming language in the ALGOL family of lexically scoped languages. In addition to the usual built-in data types, such as numbers and strings, it features both first-class hash tables and first-class functions. Although one can also program stand-alone applications in Lua, the language is mostly known for its ease of embedding. For example, Lua is widely adopted within the computer game industry [6] where it is a popular choice for scripting. Appendix A.1 provides a simplified BNF of Lua 5.1, leaving out a number of details that are inessential for this presentation. As an example, consider the following, higher-order Lua program:

```
1 function mktable(f)
2    return { x = f("x"), y = f("y") }
3 end
4
5 mktable(function (z) return z.."␣component" end)
```

The program calls a function `mktable`, passing a function as parameter. The function `mktable` will then allocate a table with two entries, `x` and `y`, initialize the entries with the result of invoking the function parameter, and return the resulting table.

2.3. *A static analysis for Lua*

To statically predict dynamic type properties of Lua programs, we build a forward, interprocedural static analysis along the lines of the TAJS type analysis for JavaScript by Jensen et al. [7]. The implementation consists of a front-end that parses a Lua program and builds a corresponding abstract syntax tree (AST). The analysis back-end consists of a composite lattice to model the state of Lua programs, as well as operations (e.g., transfer functions) over the involved lattices, and a tree-walker over the AST to model program execution.

The analysis is centered around an *allocation site abstraction* [8, 9] in which tables and function values are identified by unique labels denoting their origin. Hence, we assume that table literals and functions are uniquely labeled. Operationally, the analysis front-end takes care of such labeling.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

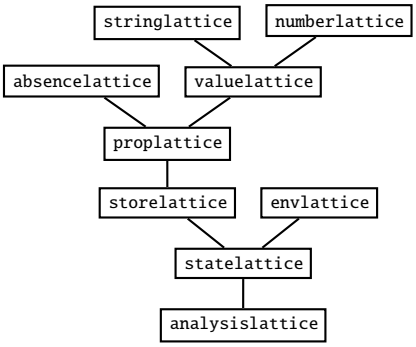

Figure 1. Diagrammatic structure of type analysis lattice.

```
module type LATTICE_TOPLESS =
sig
  type elem
  val leq       : elem -> elem -> bool
  val bot       : elem
(* val top       : elem *)
  val join      : elem -> elem -> elem
  val meet      : elem -> elem -> elem
  val to_string : elem -> string
end
```

Figure 2. Lattice signature without explicit top element.

We illustrate the composite lattice structure of the type analysis in Fig. 1. Starting from the bottom, the analysis computes invariants over `analysislattice`, which holds an abstract state (`statelattice`) for each program point (before and after each statement) of an input program. An abstract state consists of an abstract store (`storelattice`) and an abstract environment (`envlattice`) that represents scope chains. An abstract store associates to each label ℓ an element from `proplattice`, representing the properties (keys and values) of tables originating from label ℓ. Unlike JavaScript, keys in Lua tables can be any value (even tables), except the special `nil` value. The lattice `proplattice` therefore uses an additional lattice `valuelattice` to over-approximate these. The latter is a Cartesian product of `stringlattice`, `numberlattice` and a few set-based lattices to keep track of allocation sites (of tables and functions) and other value tags (e.g., Booleans and `nil`).

We express each of the above lattices as OCaml modules with a signature satisfying Fig. 2. Each of the components corresponds to an entry in the formal definition of a lattice: $\langle L; \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$. We leave an explicit top element optional, as it is not needed in practice for all lattices, for example, `valuelattice`. For lattices with an explicit top element we provide an extended lattice signature. We also include a `to_string` coercion operation in the signature for pretty printing.

As an example consider the element type of `proplattice` (abbreviated `PL`):

```
type elem =
  | Bot
  | Table of table
and table = { table       : (VL.elem * Abs.elem) TableMap.t;
              default_str : VL.elem;  (* fallback value for string keys *)
              default_key : VL.elem;  (* collective key approx. of non-string keys *)
              default     : VL.elem;  (* collective value approx. of non-string keys *)
              ... }
```

Such an element represents the first-class hash tables of Lua. In this representation `Bot` represents the empty set of hash table values and it is bound to the identifier `bot` in the `LATTICE_TOPLESS` interface

Prepared using stvrauth.cls
DOI: 10.1002/stvr

whereas the `table` type represents a set of possible hash table values. The latter internally uses an OCaml table to characterize entries with a known string key: `table` is a map from strings to pairs of `valuelattice` elements and `absencelattice` elements. We here rely on the `valuelattice` (abbreviated `VL`) to record (an approximation of) the value of each entry and we rely on the `absencelattice` (abbreviated `Abs`) to record (an approximation of) whether the entry exists (`Abs` is a two-point lattice with 'is present' and 'maybe absent' elements). Three further fields, `default_str`, `default_key`, and `default`, record (a collective approximation of) the keys and values not mentioned by the table. Entries for unknown string keys are accounted for by `default_str`. For example, for an assignment `p[s] = v` to an empty table `p` where the analysis tells us that `s` may evaluate to an unknown string ($\top$) in `stringlattice`, the analysis records the approximate value of `v` in `default_str`. Key-value entries with non-string keys are accounted for by `default_key` and `default`, respectively. Continuing our example, if instead the analysis tells us that `s` may evaluate to a number using `numberlattice`, the analysis records this approximate key in `default_key` and the approximate value of `v` in `default`.

The following operation, `find`, defined in the `proplattice` module expects a string and a table (an element of the above element type) and models a sound over-approximation of looking up the string in the table:

```
let find str map = match map with
  | Bot       -> VL.bot
  | Table map ->
    try
      let (vlat,abs) = TableMap.find (TableKey.String str) map.table in
      if abs = Abs.is_present
      then vlat
      else VL.join vlat VL.nil (* not definite read --> include nil *)
    with Not_found ->
      VL.join map.default_str VL.nil (* maybe not covered by default *)
```

Looking up a value in something that cannot be a table (`Bot` in `proplattive`) fails at runtime and hence cannot return a value (`Bot` in `valuelattice`). To look up an entry we attempt to look up the string entry in the internal OCaml table. If the `absencelattice` tells us that we can be certain that the entry exists, we simply answer with the recorded value. If we are uncertain whether the entry exists, we include in the answer the `nil` value, which models Lua's semantics of a failed table lookup. Finally, if the entry is not present in the table we fall back on the collective approximation (`default_str`) and again include `nil` to model lookup failure. Since `PL.find` is only concerned with looking up string entries it neither involves `default_key` nor `default`. Generally, `PL.find` is used to analyze table lookups, such as lvalues of the form *exp* `[` *exp* `]` (when the analysis can determine that the latter *exp* evaluates to a constant string), and method lookups in calls of the form *exp*`.`*id* `(`*exp**`)` at both the expression level and at the statement level (see Appendix A.1).

When applied to the example program of Section 2.2, our analysis will infer that the resulting store after line 5 contains a table originating (from the labeled allocation-site) in line 2. With the help of `absencelattice`, the lattice `proplattice` reveals that the allocated table definitely contains `x` and `y` entries, and `valuelattice` reveals that both entries can be any string. Because the analysis is monomorphic [1], passing two different string arguments to `f` forces the result to top in `stringlattice`.

A static analysis such as the above is typically tested on a range of hand-written programs, to ensure that the analysis soundly accounts for the corner cases of the language. This is also the situation for the present analysis. However, a number of underlying properties are seldom given similar attention. In the following sections we will develop the infrastructure for quickchecking such properties.

3. TESTING LATTICES

Formally, a lattice $\langle L; \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ satisfies a number of properties, which should be reflected in an implementation. First, $\langle L; \sqsubseteq \rangle$ is a partial order, meaning the ordering is reflexive ($\forall a \in L.\ a \sqsubseteq a$),

Prepared using stvrauth.cls
DOI: 10.1002/stvr

transitive ($\forall a,b,c \in L.\ a \sqsubseteq b \ \wedge\ b \sqsubseteq c \Rightarrow a \sqsubseteq c$), and anti-symmetric ($\forall a,b \in L.\ a \sqsubseteq b \ \wedge\ b \sqsubseteq a \Rightarrow a = b$). Second, a lattice has a range of algebraic properties:

$$\forall a \in L.\ \bot \sqsubseteq a \ \wedge\ a \sqsubseteq \top \qquad (\bot/\top \text{ is lower/upper bound})$$
$$\forall a,b \in L.\ a \sqcup b = b \sqcup a \ \wedge\ a \sqcap b = b \sqcap a \qquad (\sqcup,\sqcap \text{ commutative})$$
$$\forall a,b,c \in L.\ (a \sqcup b) \sqcup c = a \sqcup (b \sqcup c) \ \wedge\ (a \sqcap b) \sqcap c = a \sqcap (b \sqcap c) \qquad (\sqcup,\sqcap \text{ associative})$$
$$\forall a \in L.\ a \sqcup a = a \ \wedge\ a \sqcap a = a \qquad (\sqcup,\sqcap \text{ idempotent})$$
$$\forall a,b \in L.\ a \sqcup (a \sqcap b) = a \ \wedge\ a \sqcap (a \sqcup b) = a \qquad (\sqcup\text{-}\sqcap,\sqcap\text{-}\sqcup \text{ absorption})$$
$$\forall a,b \in L.\ a \sqsubseteq b \Leftrightarrow a \sqcup b = b \Leftrightarrow a \sqcap b = a \qquad (\sqsubseteq\text{-}\sqcup\text{-}\sqcap \text{ compatible})$$

In order to test such properties, we need (1) a way to compare elements for equality (e.g., to test commutativity), and (2) a scheme for generating arbitrary lattice elements. To this end, we extend the lattice signature of Fig. 2 with two additional operations for testing equality and for generating arbitrary elements:

```
val eq       : elem -> elem -> bool
val arb_elem : elem Arb.t
```

To avoid clutter and to separate our partial analysis specification from the implementation, we keep the quickchecking source code (including generators such as the above) in separate modules, on which the analysis proper does not depend. We can then achieve an "object-oriented sub-classing effect" by means of OCaml's module system: quickchecking code defines extended lattice modules that include the original lattice modules and extend their signature with additional operations, such as, `arb_elem`. In the following sections, we investigate how to formulate this operation.

3.1. Basic generators

We first consider the simple two-element lattice `absencelattice` that is used to signal whether a table entry is definitely present, in which case a table lookup is bound to succeed. Since there are only two choices of elements, an arbitrary element will necessarily be one of the two. Hence `absencelattice`'s definition of `arb_elem` coincides with that of `L` from the introduction.

For very wide lattices, such as, the flat constant propagation of strings (up to some maximum string length), the situation is more interesting. What do we mean by an arbitrary element? Potentially this could mean several things, for example: (1) a "uniform choice" where each element is equally likely to be chosen, (2) an "algebraic choice" where each datatype constructor (e.g., `Bot | Const of string | Top` in the string lattice) represents a choice reflected in the code and hence should give rise to equally likely choices, or (3) a "concretization choice" where each element is chosen based on weights reflecting how many concrete elements it represents.

For the flat constant propagation lattice, a uniform choice would mean that top and bottom are unlikely to be drawn, e.g., if we restrict the constants to, e.g., all 32-bit integers. Similarly, based on a concretization choice, top is most likely to be chosen as all concrete sets of size 2 or more abstract to it, whereas bottom is unlikely to be chosen. Hence, from a testing point of view, a distribution based on algebraic choice is preferable to uniformly cover all cases in lattice-relevant dispatches. Furthermore, this choice echoes our decision from the simpler two-element lattice.

We express the resulting generator as a choice between three simpler generators: a constant bottom generator built with `Arb.return`, the built-in string generator lifted into the `elem` type, and a constant top generator.

```
let arb_elem = Arb.(choose [return bot;
                            lift const string;
                            return top])
```

For set-based lattices, e.g., sets of allocation site labels, we need to build up a set of arbitrarily chosen elements. For this purpose, we use a fixed point combinator for generating recursive values: `fix : (base:'a Arb.t) -> ('a Arb.t -> 'a Arb.t) -> 'a Arb.t`. As a base case, we provide `LabelSet.empty`, the constructor of an empty set, suitably cast as a constant generator. As the

DOI: 10.1002/stvr

inductive case, we provide an arbitrary-set transformer, by lifting `LabelSet.add` (OCaml's built-in set addition operation) into the generators, using `arb_label` (a generator of arbitrary labels, represented as integers) and `lift2 : ('a -> 'b -> 'c) -> 'a Arb.t -> 'b Arb.t -> 'c Arb.t`:

```
Arb.(fix ~base:(return LabelSet.empty) (lift2 LabelSet.add arb_label))
```

Effectively, this generator will generate an empty `LabelSet`, then iterate an arbitrary number of times (the default maximum is 15), using `LabelSet.add` to add an arbitrary label from `arb_label` in each iteration.

Whereas these basic lattice generators are sufficient for our purposes, the approach is not limited to these lattices alone. Before we lift them to generators for composite lattices, we take a small detour to illustrate how one can also write generators for a variety of numeric abstract domains.

3.2. Numeric lattice generators

Using the same approach as in the constant propagation lattice of strings one can write a generator for a numeric constant propagation lattice [10]:

```
let arb_int  = int_range ~start:min_int ~stop:max_int
let arb_elem = Arb.(choose [return Bot;
                    lift (fun i -> Const i) arb_int;
                    return Top])
```

Here we first formulate a generator of integers `arb_int` which may generate all constants between OCaml's least representable number and the greatest representable number (the bitwidth depends on the underlying machine's architecture) and subsequently lift it to help generate constants. Another option would be to use the built-in `int` combinator that generates numbers between 0 and its supplied argument.

As another example of a simple numeric domain we consider the parity lattice. We can represent the elements of the parity domain as an algebraic data type: `type elem = Bot | Even | Odd | Top`. Based on this representation we can then write a corresponding generator that selects uniformly between its four elements:

```
let arb_elem = Arb.among [Bot; Even; Odd; Top]
```

Similarly we can represent a simple sign domain as an algebraic data type with five constructors: `type elem = Bot | Neg | Zero | Pos | Top` and write a corresponding generator that chooses uniformly between them:

```
let arb_elem = Arb.among [Bot; Neg; Zero; Pos; Top]
```

Intervals [11] is a classical example of a numeric abstract domain. One can represent intervals in multiple ways. One representation that directly matches the formalization (from Cousot and Cousot [12])

$$I = \{[l;u] \mid l \in \mathbb{Z} \cup \{-\infty\} \wedge u \in \mathbb{Z} \cup \{+\infty\} \wedge l \leq u\} \cup \{\bot\}$$

is as three algebraic data types—one for lower bounds, one for upper bounds, and one for intervals:

```
type lowerbound = MInf | LBound of int
type upperbound = PInf | UBound of int
type elem       = Bot | Interval of lowerbound * upperbound (* [l;u] where l <= u *)
```

Furthermore we can maintain the invariant $l \leq u$ by writing a *smart constructor* [13]: a function conveniently named `interval` that takes a pair of lower/upper bounds and maps empty ranges ($l > u$) to `Bot` and non-empty ranges to applications of the `Interval` constructor. With this representation in mind we can now write a generator for intervals by combining two simpler generators for lower bounds and upper bounds:

```
let arb_lowerbound = Arb.(choose [return MInf;
                                  lift (fun i -> LBound i) arb_int])
let arb_upperbound = Arb.(choose [return PInf;
                                  lift (fun i -> UBound i) arb_int])
let arb_elem = Arb.lift2 (fun l u -> interval (l,u)) arb_lowerbound arb_upperbound
```


DOI: 10.1002/stvr

Note how we have chosen not to generate `Bot` elements directly. However, our use of the smart constructor `interval` ensures that we still generate `Bot` elements indirectly when the bounds satisfy $l > u$. One may choose to adjust the bound generators to generate the extremal values `MInf` and `PInf` less often, e.g., with a 25% chance. To increase the chance of generating non-bottom elements one could alternatively swap two non-extreme bounds l and u when $l > u$ and fix the chance of generating a bottom element to, e.g., 10%. We will revisit our interval generation strategies in Section 3.4.

As a final example of a numeric abstract domain we consider congruences [14]. Elements of the congruence domain are of the form $a + b\mathbb{Z}$ where both the modulo b and the remainder a are simultaneously discovered. As usual in modulo arithmetic we expect the invariant $0 \leq a < b$ to hold. Furthermore, congruence elements of the form $a + 0\mathbb{Z}$ (where $b = 0$) can express the numeric constant propagation lattice, thereby giving rise to the data type invariant $b = 0 \ \vee \ 0 \leq a < b$. We can now represent congruences as a two element algebraic data type:

```
type elem = Bot | Cong of int * int (* (a,b)  where  b=0  or  0 <= a < b *)
```

Again we can write a smart constructor `cong` to help maintain the above data type invariant by letting `cong`(a,b) normalize the first argument to a mod b when $b \neq 0$ and $a \geq b$. With these in mind we can now formulate a generator of arbitrary congruence elements by lifting two integer generators:

```
let arb_elem = Arb.(choose [return Bot;
                            lift2 (fun a b -> cong (a,b)) (int 10) (int 10);])
```

Here we have chosen to generate modulo and remainder between 0 and 10 as an initial stress test of the logic behind the representation. We do so since the chance of hitting (and uncovering errors in) the branches concerning $b = 0$ get smaller as the range of a and b increases. Alternatively one could choose to write a custom integer generator covering a larger range, yet with a reasonable chance (e.g., 5 or 10%) of generating extremal values such as 0. Since `cong`'s normalization of the first argument skews the test distribution towards situations where a internally is in the lower half of the range $[0; b-1]$, one may choose to supplement the above with a generator that first generates b and then subsequently generates a in the above interval when $b \neq 0$.

With generators in place for a range of basic and numeric lattices, we now turn to lifting them to generators for composite lattices.

3.3. Composite generators

We can easily form generators for product lattices by composing the generators of the sub-lattices. For example, if `A` and `B` are lattice modules extended with generators, we can form a generator for the pair lattice of elements `A.elem * B.elem`:

```
let arb_elem = Arb.pair A.arb_elem B.arb_elem
```

Concretely, we use this approach to form a generator for `statelattice` (represented as two-element records) of the generators for `storelattice` and `envlattice`.

The approach extends to *reduced products* as well: We have already seen how to formulate an interval generator out of two generators for the lower and upper bounds, respectively. Intervals are in fact a reduced product of a maximum abstraction and a minimum abstraction, in which our smart constructor takes care of reducing the constructed elements.

To build arbitrary elements of function lattices, such as `storelattice` and `analysislattice`, we first formulate a helper for building maps. The helper takes three arguments: `mt` for building the empty map, `add` for adding arguments, and finally an association list `kvs` of (key, value) pairs. It can be implemented as a simple fold over the input list, utilizing `mt` in the base case and subsequently adding each (key, value) pair using `add`:

```
let build_map mt add ls = List.fold_right (fun (k,v) acctbl -> add k v acctbl) ls mt
```

Maps are built based on the input list's element order. If `build_map` is applied to the same association list twice, albeit with the elements permuted, the resulting map (and hence OCaml's underlying balanced tree) will likely result in a differently structured tree, thereby avoiding the

DOI: 10.1002/stvr

pitfall of skewing the generator into generating only a particular subset of map shapes. Any such skewing will affect the quality of our randomized testing, and potentially let buggy code go through unnoticed [15]. We can now generate arbitrary maps. For `storelattice`, e.g., we utilize `Arb.map : 'a Arb.t -> ('a -> 'b) -> 'b Arb.t` to first form a generator of arbitrary (`label`, `proplattice`) association lists and subsequently transform the outcome using `build_storemap`, which specializes `build_map` to build `StoreMaps`:

```
let arb_entries    = Arb.(list ~len:(int 20) (pair arb_label pl_arb_elem))
let build_storemap = build_map StoreMap.empty StoreMap.add
let arb_elem       = Arb.map arb_entries build_storemap
```

To summarize, we have seen how to build (and combine) generators for a simple two-element lattice, a constant propagation lattice, a set-based lattice, product lattices, and function lattices. Collectively, these lattices can be combined to a full-scale static type analysis such as TAJS [7]. In addition we have seen how to write generators for numeric lattices.

3.4. Testing lattice properties

With element generators in place for all lattices, we are now in position to check the lattice properties we set out to. The lattice property tests are furthermore independent of the particular lattice at hand as long as the lattice satisfies the `LATTICE_TOPLESS` signature. As such, we can wrap the tests in a reusable functor `GenericTests`.[†] For example, we can formulate a generic join commutativity test for any lattice `L` that satisfies the `LATTICE_TOPLESS` signature:

```
let join_comm = (* forall a,b. a \/ b = b \/ a *)
  mk_test ~n:1000 ~pp:pp_pair ~name:("join␣commutative␣in␣" ^ L.name)
    arb_pair (fun (a,b) -> L.(eq (join a b) (join b a)))
```

where `arb_pair` and `pp_pair` are defined as in Section 2.1. We can subsequently test our example lattices for commutativity of their join operations.

Consider now a conditional property, such as transitivity of the lattice ordering. Again this translates directly to a test:

```
(* forall a,b,c. a <= b /\ b <= c  =>  a <= c *)
let leq_trans =
  mk_test ~n:1000 ~pp:pp_triple ~name:("leq␣transitive␣in␣" ^ L.name)
    arb_triple (fun (a,b,c) -> Prop.assume (L.leq a b);
                               Prop.assume (L.leq b c);
                               L.leq a c)
```

Here, `arb_triple` and `pp_triple` are generic helper functions for generating and pretty-printing arbitrary triples, analogous to `arb_pair` and `pp_pair`. This approach is insufficient for more complex lattices, however, as the probability of generating arbitrary triples that are ordered (and thus satisfying the precondition) decreases with the number of lattice elements. For example, if we run the above test on 1000 arbitrary generated input triples of `valuelattice` elements we see a problem:

```
testing property leq transitive in value lattice...
  [✓] passed 1000 tests (1000 preconditions failed)
```

Not a single generated triple satisfies the precondition.

Rather than cranking up the number of generated triples to increase the chance of generating a few ordered ones we instead equip lattices with a generator to help generate ordered tuples. To this end, we further extend the lattice signature of Fig. 2 with an operation for generating arbitrary elements less or equal to a given argument:

```
val arb_elem_le : elem -> elem Arb.t
```

The two-element `absencelattice` is straightforward to extend:

```
let arb_elem_le e = if e = Top then arb_elem else Arb.return Bot
```

[†]The functor's signature is available in Appendix A.2 and further discussed in Section 5.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

The extension to the flat `stringlattice` is not considerably more complex:

```
let arb_elem_le e = match e with
  | Bot     -> Arb.return Bot
  | Const s -> Arb.among [Bot; Const s]
  | Top     -> arb_elem
```

To build an arbitrary subset of a given set, which we need for generating ordered tuples involving set-based lattices, we first formulate a helper function `build_set` akin to `build_map`:

```
let rec build_set mt sglton union ls = match ls with
  | []  -> Arb.return mt
  | [l] -> Arb.return (sglton l)
  | _   -> Arb.(int (1 + List.length ls) >>= fun i ->
                let ls,rs = split i ls in
                lift2 union (build_set mt sglton union ls)
                            (build_set mt sglton union rs))
```

Similarly to `build_map`, `build_set` is parameterized with a builder for the empty set, a builder for singletons, a set union operation, and a list of elements. For input lists of length two or more, `build_set` will split the input list at some arbitrary point, recurse on both halves, and union their results, again to avoid the pitfall of generating skewed data structures. Note how this case utilizes `(>>=) : 'a Arb.t -> ('a -> 'b Arb.t) -> 'b`, the monadic bind of generators, to temporarily name the chosen index for splitting.

Next, we formulate a helper function `le_gen` to aid with subset selection. The function is parameterized with a list and a builder. It will first permute its input list, split the resulting list in two sublists, and finally pass one of these to the builder:

```
let le_gen es build =
  let es_gen = permute es in
  Arb.(es_gen >>= fun es ->
       int (1 + List.length es) >>= fun i ->
       let smaller_es,_ = split i es in
       build smaller_es)
```

Within `valuelattice` we use this approach repeatedly to generate subsets of its set-based lattices. One of `valuelattice`'s set-based lattices records a set of runtime tags that a value can have, represented as a `TagSet`, which is a built-in set-type specialized to contain only `tag` elements. For example, given an argument `e` from `valuelattice`, we serialize its tags into a list using `TagSet.elements`, and pass the result to `le_gen` for subset selection and subsequent building of a set structure:

```
let build_tagset = build_set TagSet.empty TagSet.singleton TagSet.union
let le_tag_gen   = le_gen (TagSet.elements e.tags) build_tagset
```

Assuming the outputs of `permute` and `Arb.int` are arbitrary, this approach provides equal chance of each set size. Alternatively, one could consider an approach with equal chance of each subset, by flipping a coin to decide whether each element is included in the resulting subset.

Regarding the numeric domains, we can reuse the approach from the flat `stringlattice` to write `arb_elem_le` for a numeric constant propagation lattice. Writing `arb_elem_le` for the parity and sign lattices is not much harder. For example, a version for the parity lattice is the following:

```
let arb_elem_le e = match e with
  | Bot  -> Arb.return Bot
  | Even -> Arb.among [Bot; Even]
  | Odd  -> Arb.among [Bot; Odd]
  | Top  -> arb_elem
```

To define `arb_elem_le` for the interval lattice we continue the type-directed approach and formulate generators of bound elements for both lower bounds and upper bounds.[‡] With these in hand we can now formulate `arb_elem_le` as a straightforward combination:

[‡]Technically, `int_range` generates integers less than the given `stop` parameter in `arb_int`, which prevents `(u+1)` in `arb_upperbound_le` from overflowing.

DOI: 10.1002/stvr

```
let arb_lowerbound_ge l = match l with
  | MInf     -> arb_lowerbound
  | LBound l -> Arb.(lift (fun l' -> LBound l') (int_range ~start:l ~stop:max_int))
let arb_upperbound_le ub = match u with
  | PInf     -> arb_upperbound
  | UBound u -> Arb.(lift (fun u' -> UBound u') (int_range ~start:min_int ~stop:(u+1)))
let arb_elem_le e  = match e with
  | Bot            -> Arb.return Bot
  | Interval (l,u) -> Arb.lift2 (fun l' u' -> interval (l',u'))
                                (arb_lowerbound_ge l) (arb_upperbound_le u)
```

Once again our use of the smart constructor `interval` may reduce the generated element to `Bot`. This approach has only a small chance of generating an interval equal to the given argument, which happens when both `arb_lowerbound_ge` and `arb_upperbound_le` generate equal arguments. Such a corner case could in principle be a source of errors. As an alternative one could therefore let the last branch choose between three generators: one that may increase the lower bound, one that may decrease the upper bound, and one that may change both (as above).

Finally, we can formulate `arb_elem_le` also for the congruence domain:

```
let arb_elem_le e  = match e with
  | Bot        -> Arb.return Bot
  | Cong (0,1) -> arb_elem
  | Cong (a,b) ->
      Arb.(choose [return Bot;
                   lift (fun x -> cong (a+b*x,0)) (int 10);
                   lift2 (fun x b' -> cong (a+b*x,lcm b b')) (int 10) (int 10)])
```

In the second case, the element $0+1\mathbb{Z}$ in the numeric congruence domain represents $\mathbb{Z}$ itself—the top of the lattice: any element is therefore less or equal to it. As to the last case, it is a choice between three generators:

(a) a constant generator of `Bot` which is less or equal to any element,
(b) a generator of elements of the form $r+0\mathbb{Z}$ representing constants (e.g., given a parameter $3+7\mathbb{Z}$ representing $\{3,10,17,24,31,38,\ldots\}$ this choice will pick an arbitrary integer, e.g., 4, and generate $(3+7*4)+0\mathbb{Z}=31+0\mathbb{Z}$ which represents $\{31\}$ and is strictly less than $3+7\mathbb{Z}$ in the lattice order), and
(c) a generator of a larger modulo with a compatible remainder (e.g., given a parameter $3+7\mathbb{Z}$ we pick two arbitrary numbers, e.g., 2 and 5, we compute a new modulo as the *least common multiple* of 7 and 5: `lcm 7 5` $=35$, and we compute a new remainder: $3+7*2=17$, thereby giving rise to $17+35\mathbb{Z}$, which represents $\{17,52,\ldots\}$, again strictly less than $3+7\mathbb{Z}$).

As to the composite lattices, formulating `arb_elem_le` for product lattices is a straightforward lifting that generates and combines less-or-equal elements for each sub-lattice. Such a formulation also suffers from our interval generator's drawback: it has only a small chance of generating a pair equal to its argument. One may therefore wish to choose between three generators as suggested for the intervals. For function lattices under pointwise ordering, we first serialize its bindings into an association list. We then reuse the `le_gen` function from above to choose a subset of bindings. This has the effect of choosing fairly between each subset size of bindings. (Alternatively we could have used a coin toss per binding, similar to above.) We then iterate over the resulting association list using `le_entries` below, which invokes its argument `arb_elem_le` on each entry in order to obtain a result that may be pointwise less than its argument. Finally we transform the resulting association list with `build_map`.

```
let le_entries arb_elem_le kvs =
  let rec build es = match es with
    | []       -> Arb.return []
    | (k,v)::es -> Arb.(build es >>= fun es' ->
                        arb_elem_le v >>= fun v' ->
                        return ((k,v')::es')) in
  build kvs
```


DOI: 10.1002/stvr

With `arb_elem_le` in hand, we can now generate arbitrary ordered pairs and triples for any lattice `L` to test, e.g., transitivity. For example, we can define `ord_pair` as follows:

```
let ord_pair = Arb.(L.arb_elem >>= fun e -> pair (L.arb_elem_le e) (return e))
```

Alternatively, one could have chosen to extend lattice signatures with an `arb_elem_ge` operation for generating lattice elements *greater than or equal* to a given element. Generating ordered pairs with such an approach would skew generation towards larger second components, whereas the current approach based on `arb_elem_le` skews the generation towards smaller first components. Ideally, one should therefore expand lattice signatures with both operations and let `ord_pair` flip a coin to decide between the two approaches. As this would require writing more generator code we have not pursued this option further.

As a consistency check, we include in our generic test suite a quickcheck test that generates ordered pairs `(a,b)` and tests that they are in fact ordered according to the `leq` ordering:

```
let check_ordering =
  mk_test ~n:1000 ~pp:pp_pair ~name:("ordered␣pairs␣consistent␣in␣" ^ L.name) ...
    ord_pair (fun (a,b) -> L.leq a b)
```

For every lattice module `L` that satisfies the `LATTICE_TOPLESS` signature, we test that $\bot$ is a lower bound:

```
let bot_is_lowerbound = (* forall a. bot <= a *)
  mk_test ~n:1000 ~pp:L.to_string ~name:("bot␣is␣lower␣bound␣in␣" ^ L.name) ...
    L.arb_elem (fun a -> L.(leq bot a))
```

Similarly for lattice modules with an explicit top element, we test whether all elements are less than or equal to it. Another property one may consider is that in every lattice, $\bot$ should be the *greatest* lower bound of all the elements in L ($\bot = \sqcap L$) and $\top$ should similarly be the *least* upper bound in lattices with an explicit $\top$ element ($\top = \sqcup L$). However, in general it is not tractable to generate a list of all lattice elements, e.g., for the interval lattice over 32-bit numbers, to test these properties and we therefore make no attempt to do so.

4. TESTING LATTICE OPERATIONS

We now turn to operations on lattices, such as `PL.find`. Monotonicity, $x \sqsubseteq x' \implies f(x) \sqsubseteq f(x')$, of functions operating over lattices is central to static analyses due to Tarski's fixed point theorem. This property furthermore lends itself to quickchecking: generate two arbitrary, ordered elements, apply the operator f to both, and test the resulting values for ordering. Alternatively we could have expressed monotonicity as $f(x) \sqcup f(x') \sqsubseteq f(x \sqcup x')$ or even $f(x \sqcap x') \sqsubseteq f(x) \sqcap f(x')$ as both of these are equivalent to the above definition, mathematically speaking. Each of these would lend themselves to a different testing strategy, that involves generating two arbitrary elements, applying the operator f three times, computing two joins (or meets), and performing a final ordering test. Depending on the quality of our generators, each of these approaches could potentially exercise other paths in a static analysis and thereby discover other errors.

What other properties are desirable of an analysis operator? Strictness, $f(\bot) = \bot$, is an obvious candidate. Depending on the lattices, this can mean "no output values produced when given no input values" or "output state is unreachable if input state is unreachable". Since we are only interested in sound analyses, operator strictness is not a formal requirement: returning any over-approximation of $\bot$ is safe, yet an analysis should be as precise as possible. As such, testing a given analysis implementation for strictness can help detect opportunities for precision improvements.

As a third operator property, we include invariance (also known as congruence), $\forall x, x'.\ x = x' \Rightarrow f(x) = f(x')$: operators should yield equal results when applied to equal arguments. Mathematically speaking, this should be obvious, yet it is less so in an implementation, where, e.g., identical strings may be located at different places in memory, or, e.g., a set data structure containing the same elements may be differently shaped depending on the insertion order. In day-to-day programming, this manifests itself as errors associated to confusing reference (pointer) equality

Prepared using stvrauth.cls
DOI: 10.1002/stvr

```
(* forall s. bot = s ^ bot *)
let concat_strict_snd =
  mk_test ~n:1000 ~pp:Str.to_string
    ~name:("concat␣strict␣in␣argument␣2")
    Str.arb_elem (fun s -> Str.(eq bot (concat s bot)))

(* forall s,s',s''. s' <= s''  =>  (s ^ s') <= (s ^ s'') *)
let concat_monotone_snd =
  mk_test ~n:1000 ~pp:(PP.pair Str.to_string pp_pair)
     ~name:("concat␣monotone␣in␣argument␣2")
    (Arb.pair Str.arb_elem ord_pair)
    (fun (s,(s',s'')) -> Prop.assume (Str.leq s' s'');
                         Str.(leq (concat s s') (concat s s'')))

(* forall s,s',s''. s' ~ s''  =>  (s ^ s') ~ (s ^ s'') *)
let concat_invariant_snd =
  mk_test ~n:1000 ~pp:(PP.pair Str.to_string pp_pair)
    ~name:("concat␣invariant␣in␣argument␣2")
    (Arb.pair Str.arb_elem Str.equiv_pair)
    (fun (s,(s',s'')) -> Prop.assume (Str.eq s' s'');
                         Str.(eq (concat s s') (concat s s'')))
```

Figure 3. Examples of specific `stringlattice` operation tests.

with structural equality or hand-written equality predicates. In the broader context of quickchecking, Holdermans [15] has further argued for supplementing "axiom driven tests" with invariance tests to help catch an otherwise undiscovered class of errors (this issue is discussed more in Section 6). In order to test for invariance, we extend the lattice signature with a generator of equivalent element pairs:

```
val equiv_pair : (elem * elem) Arb.t
```

One must then again go through the lattices to extend them with such a generator. For example, for `stringlattice`, our generator uses OCaml's built-in `String.copy` to create pairs of equivalent, yet differently located strings. For set lattices, `build_set` is already geared to create potentially differently shaped trees for each invocation. For function lattices, extra care must be taken to avoid that duplicate key entries in the initial association list will result in a different function lattice element under a different addition order. We do so by first building one map, then serialize its bindings into an association list, which we can subsequently permute and use to build a second, equivalent map.

As a fourth operator property we include distributivity, $\forall x,x'.\ f(x \sqcup x') = f(x) \sqcup f(x')$. The primary motivation for considering this property is analysis speed: a static analysis where the transfer functions are distributive can sometimes be implemented with faster analysis algorithms [1]. The ability to test for distributivity should therefore be welcome to static analysis developers.

Returning to the topic of lattice operations, consider the tests in Fig. 3 related to strictness, monotonicity, and invariance of the `stringlattice` (`Str`) operation `concat` that conservatively models concatenation of strings. By studying Fig. 3, one can observe a pattern in the code for the three tests, to the point that it is needlessly repetitive. To avoid writing such repetitive lines, we seek to distill a basis of primitives (an EDSL) from which all such tests can be expressed concisely.

The EDSL consists of primitives for expressing properties of unary operations: `op_strict`, `op_monotone`, `op_invariant`, and `op_distributive`. In addition, we add generic operations, `pw_left` and `pw_right` for adding arguments to the left, resp. right, of the parameter in question. Finally, we add a generic operator named `finalize` for building a test out of the pieces, effectively sealing off the signature (we omit a few optional parameters to `mk_test`):

```
let finalize opsig (opname,op) =
  opsig
    (fun (pp,gen,prop,pname,leftargs) ->
       mk_test ~n:1000 ~pp:pp (* ... *)
         ~name:(Printf.sprintf "'%s␣%s␣in␣argument␣%i'" opname pname leftargs)
         gen (prop op))
```


DOI: 10.1002/stvr

```
mname    ::= (module NAME)
baseprop ::= op_monotone
           | op_strict
           | op_invariant
           | op_distributive
rightprop ::= baseprop
            | pw_right mname (rightprop)
leftprop ::= rightprop
           | pw_left mname (leftprop)
opsig    ::= (leftprop mname mname)
prop     ::= finalize opsig
```

(a) Combinator-based notation.

```
mname    ::= (module NAME)
baseprop ::= mname -<-> mname
           | mname -$-> mname
           | mname -~-> mname
           | mname -%-> mname
opsig    ::= [mname ---> ]* baseprop [---> mname]*
prop     ::= (testsig opsig) for_op
```

(b) Infix notation.

Figure 4. EBNF grammar of our EDSL.

This definition hints to the implementation of our framework: the combinators traverse the signature description in continuation-passing style (CPS). In doing so, they will structurally build up a pretty-printer `pp`, a generator `gen`, the property `prop`, the property name `pname`, and a left argument counter `leftargs`. When the traversal is complete, all the elements are in place for `finalize` to supply an initial continuation, representing the last thing to be done: We create a test with `mk_test` and pass it the created pretty printer, generator and (parameterized) property. By further passing the operator name `opname` as a string and the number of left arguments, we can thus obtain nice error messages (as illustrated in Section 1). For example, we can express the monotonicity test corresponding to `concat_monotone_snd` in Fig. 3, based solely on a description of the signature:

```
# let str_concat = ("Str.concat",Str.concat) in
  finalize (pw_left (module Str) op_monotone (module Str) (module Str)) str_concat;;
- : QCheck.test = <abstr>
```

where in `(module Str)` we utilize OCaml's support for first-class modules to express each lattice module of the signature. By further utilizing OCaml's infix syntax to define `=::` as a shorthand for `finalize` we can express the same test as a one-liner:

```
pw_left (module Str) op_monotone (module Str) (module Str) =:: ("Str.concat",Str.concat)
```

Because of the embedding into OCaml, the EDSL will furthermore statically ensure that each module satisfies the lattice signature and that the described signature and the operator's signature agree. In Fig. 4(a) we summarize the syntax of the combinator-based EDSL and in Appendix A.2 we summarize its signatures.

The above description may still not be quite satisfactory, as the connection to the corresponding mathematical notation $Str \longrightarrow Str \xrightarrow{\sqsubseteq} Str$ is unclear. To remedy this mismatch we provide convenient infix syntax in the form of arrows with and without annotation: `-$->` for strictness, `-<->` for monotonicity, `-~->` for invariance, `-%->` for distributivity, and `--->` for a plain function arrow. With the infix arrow syntax, we can now express a monotonicity test corresponding to `concat_monotone_snd` in Fig. 3 based almost exclusively on `Str.concat`'s signature:

```
# (testsig (module Str) ---> (module Str) -<-> (module Str)) for_op;;
- : string * (Str.elem -> Str.elem -> Str.elem) -> QCheck.test
  = <fun>
```

where `-<->` marks that we wish to test the second parameter of the signature for monotonicity and `testsig` and `for_op` act as delimiters of the signature. Note how OCaml's type checker infers that the EDSL expression further expects a string (describing the operator, e.g., by name) and a `Str.elem -> Str.elem -> Str.elem` operator to yield a quickcheck test. We thereby statically prevent type errors in our tests, e.g., an attempt to quickcheck a lattice operator over an incorrect signature.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

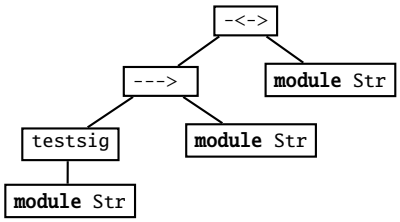

Figure 5. OCaml's underlying AST.

Similarly, we can concisely test for distributivity. For example, it is well known that the squaring operator ($f(x) = x * x$) over the numeric sign domain is not distributive [1]. A counterexample is immediately found:

```
# let square_distributive =
    (testsig (module Sign) -%-> (module Sign)) for_op ("square", fun s -> Sign.mult s s) in
  run square_distributive;;
testing property 'square distributive in argument 1'...
  [✗] 77 failures over 1000 (print at most 1):
  (Neg, Pos)
```

Indeed, squaring both a negative and a positive number results in a positive number, while the lattice join of the two yields $\top$ (any number) and squaring any number can yield any number.

Technically, the type-safe embedding is achieved by left-associativity of function application and of all infix operators in OCaml beginning with the minus '`-`' character. As a result, OCaml itself will create an underlying AST in the form of Fig. 5. By suitable function definitions for the `testsig`, `for_op`, and the infix arrow operators, one can obtain a depth-first traversal compatible with an AST such as the one in Fig. 5. The challenge is to combine the combinators in such a traversal: we do not know the arguments to, e.g., `op_monotone` until we meet the `-<->` node (the rightmost `Str` in its left subtree and the right child of the root). Furthermore, the role of additional arguments changes above and below such a *baseprop* node: below they should be added as left arguments with the combinator `pw_left`, and above they should be added as right arguments with the combinator `pw_right`. We solve these issues by implementing the traversal as a state machine. One register of the state machine keeps track of the latest encountered module argument (initialized with `testsig`'s argument). Two other state machine registers accumulate both a left- and a right argument-adding transformer simultaneously. Upon meeting an `--->` arrow, we register the latest module argument and add the previous to both argument-adding transformers. Upon meeting a '*baseprop* arrow', such as the `-<->` node, all previously visited arguments were to the left, so we move the left transformer to the right transformer for any remaining right arguments to be added. Upon completion, a full argument-adding transformer is therefore available to `for_op`. We refer the interested reader to the source code of our library for more details on the type-safe embedding.

Statically typing something of this form may seem hard, as the type of a signature varies with its shape. The situation is similar to the static typing of C's `printf`, which will vary with its format string. Inspired by Danvy's solution to the `printf` problem [16], we utilize the polymorphism of result types in continuation-passing style. In this context, `for_op` represents the initial continuation and takes care of instantiating the polymorphic result type. It is implemented as a call to `finalize`. The EDSL's infix syntax is summarized in Fig. 4(b). With `=:` as an additional infix synonym for `for_op`, we can thus write the equivalent tests of Fig. 3 as compactly as follows:

```
let concat_tests =
 let str_concat = ("Str.concat",Str.concat) in [
 testsig (module Str) ---> (module Str) -$-> (module Str) =: str_concat;
 testsig (module Str) ---> (module Str) -<-> (module Str) =: str_concat;
 testsig (module Str) ---> (module Str) -~-> (module Str) =: str_concat;
 (* ... and similar for the first argument ... *) ]
```


DOI: 10.1002/stvr

```
module type ARB_ARG =
sig
  type elem
  val arb_elem  : elem Arb.t
  val to_string : elem -> string
end
```

Figure 6. Relaxed argument signature.

4.1. Checking predicates

A number of lattice operations naturally involve only lattice arguments. However, many operations have a different signature. One such class is predicate functions with a Boolean result type. For example, `VL.may_be_proc : VL.elem -> bool` takes a `valuelattice` (`VL`) element and checks if the set-lattice that over-approximates (allocation sites of) procedure values is nonempty. In fact, this is still an operator on lattices, as the Booleans form a lattice, whose elements are ordered by implication, and `VL.may_be_proc` is monotone over this lattice. For this purpose, we have implemented a short Boolean lattice module, tested the lattice implementation (using quickchecking, of course), and finally quickchecked `VL.may_be_proc` and a number of similar queries using the Boolean lattice. For other queries, e.g., `VL.is_bot`, which checks whether a given `valuelattice` element is bottom, we use the dual Boolean lattice that is ordered by reverse implication.

We cannot expect the arguments of all operations to have been designed as lattices. Some arguments and results nevertheless turn out to be so in retrospect. Rather than forcing developers to revise their analysis implementations, we also provide OCaml functors to easily build, e.g., list and pair lattices for such quickchecking. For example, the function `find_exn` in `proplattice` returns a pair consisting of a `valuelattice` element and an `absencelattice` element. To test it using our approach, it is therefore convenient to instantiate a "value and absence pair" lattice `VLAbsPair` purely for testing purposes and use `VLAbsPair` in the signature-based tests of `find_exn`.

4.2. Beyond pure lattice operations

Our EDSL of infix signature syntax between modules satisfying `LATTICE_TOPLESS` can handle a majority of the operations in our Lua analysis implementation. However, the requirement that arguments must satisfy the `LATTICE_TOPLESS` signature is sometimes too restrictive: some operations simply accept an allocation site label or a `string` (representing a variable name or a property), yet can still be strict, monotone, and/or invariant in the other arguments.

To handle such cases, we relax the parameter requirements to the `pw_left` and `pw_right` combinators, to match the simpler `ARB_ARG` signature in Fig. 6. Based on this relaxation we can then write the last tests in the (still type-safe, but less readable) combinator syntax. We have not found a type-safe approach to allow these in the infix syntax at this point. Doing so would require a context-sensitive typing of the state-machine implementation: the register containing the latest encountered module argument should then either hold a module satisfying the `ARB_ARG` or `LATTICE_TOPLESS` signatures, depending on the context.

An avenue for a different, non-signature based class of tests is that of soundness. To a limited degree this is feasible. For example, we include the following string concatenation test over the `stringlattice` in our test suite.

```
(* forall s,s'. abs(s ^ s') = abs(s) ^ abs(s') *)
let concat_sound =
  mk_test ~n:1000 ~pp:pp_pair ~name:("Str.concat␣sound")
    Arb.(pair string string)
    (fun (s,s') -> Str.(eq (const (s ^ s')) (concat (const s) (const s'))))
```

This, however, will quickly require a reference interpreter in some form as well as the ability to generate arbitrary syntax trees. We leave such an exploration for future work.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

5. EXPERIMENTS AND DISCUSSION

To test our hypothesis that quickchecking a static analysis increases confidence in its implementation, we consider the following two research questions: Can quickchecking expose basic properties and detect errors in static analysis implementations? How does code coverage of quickchecking compare with a traditional test suite? We focus on universal properties that relate to lattices in general, not to the specific analysis. We address these questions by reporting on the lessons learned from our Lua analysis case study.

Errors found by quickchecking Indeed, we have found errors in our initial Lua analysis implementation using the lattice property tests and using the lattice operation tests. For later extensions of the analysis, quickchecking became a natural part of the development cycle. As an example, we found an early copy-paste error in the implementation of `meet` of `absencelattice`. This error was caught early because `meet` failed the algebraic tests. We have found several lattice operations that were not strict but could be improved to be so. For example, the function `unop` in `valuelattice` accepting a unary operation and an element of `valuelattice` and again producing a `valuelattice` was initially implemented as follows:

```
let unop op lat = match op with
  | Ast.Uminus ->
    let lat' = coerce_tonum lat in
    if may_be_number lat' (* unary minus of number (or better) is number *)
    then number
    else bot
  | Ast.Length ->
    if may_be_strings lat || may_be_table lat
    then number
    else bot    (* length of everything but strings and tables is number *)
  | Ast.Not    ->
    bool   (* negation of anything is bool *)
```

This implementation is not strict in the second parameter as it returns a non-bottom result when applied to, e.g., arguments `Ast.Not` and `bot` from `valuelattice`. The function was easily made strict by initially testing `lat` for `bot` and returning `bot` early and only proceeding with the above dispatch when the test fails. As such, quickchecking helped make our analysis more precise. More importantly, we found two (unrelated) operations that were not monotone even though they were supposed to be. At closer inspection, the issue turned out to be a lack of relational coordination across lattices: operations would iterate over labels found in the `valuelattice`, yet some labels would not exist as entries in the `storelattice` and hence looking them up would fail, ultimately leading to non-monotonicity. Even though such issues represent corner cases and perhaps never occur in a proper analysis execution, having the code act meaningfully is preferable.

An in-depth story of quickchecking `proplattice` Initially, our design of `proplattice` had an internal OCaml table to keep track of all statically known entries, and a default entry to over-approximate both all unknown entries with string keys (as in TAJS [7]) as well as entries with non-string keys:

```
type elem = { table       : (VL.elem * Abs.elem) TableMap.t;
              default     : VL.elem;
              ... }
```

This design leads to a crude over-approximation for non-string lookups: conservatively, we would have to return the value of `default`, e.g., when looking up a Boolean even though perhaps only number keys had been written to a table. As a consequence, we refined the design to include a collective over-approximation of all unknown keys:

```
              default_key : VL.elem;
```

With this refined design, we could compare the key of a potentially absent non-string entry, and precisely model lookup failure if the key was not less than or equal to `default_key` under the `valuelattice` ordering. Continuing our example from above, since a `valuelattice` value containing

DOI: 10.1002/stvr

only a singleton `TagSet` with `Bool` would not be less than or equal to the `valuelattice` value with top in `numberlattice`, we would know the lookup would fail.

To decide (partial) ordering between two `proplattice` values `p` and `p'`, the operation `leq` would check the ordering of both pairs of defaults: `p.default_key` against `p'.default_key` and `p.default` against `p'.default`, as well as check that each statically known entry of `p.table` would be accounted for in `p'` by a similar entry in `p'.table` or by the default entries. To compute the join (the least upper bound) of two `proplattice` values `p` and `p'`, we would combine the joins of their default entries with an iteration over their statically known table entries. For example, a statically known entry `"a"` from `p.table` would be included in the resulting table with either the same entry or with a entry that joins the original with `p'.default` if `"a"` was accounted for by `p'.default_key` (and vice versa).

In order to quickcheck the `proplattice` lattice, our `arb_elem` function would generate and compose arbitrary elements for each of these subparts, using `VL.arb_elem` for the default entries and the recipe for function lattices for the internal table. Similarly, `arb_elem_le` would generate and compose arbitrary elements less than a given `proplattice` value for each of these subparts. This design formed the basis of the type analysis described in the conference version of this paper [3].

As part of a recent revision we refined our quickcheck generators, as suggested by a reviewer. Since analysis code will sometimes look up in a `storelattice` the allocation-site labels (of functional values and records) occurring in `valuelattice` values, we refined the generator to either (a) generate a label among a small fixed set of labels (integers $0, \ldots, 7$) or (b) to fall back on one of `qcheck`'s built-in integer generators. Similarly, since analysis code will sometimes use the approximate strings of a `stringlattice` to index a `proplattice` value, we refined the generator of arbitrary strings (used by both lattices) to either generate a string among a small fixed set of strings (`["a";"b";...]`) or fall back on `qcheck`'s built-in string generator. This approach can obviously be refined even further, e.g., by initially generating a small random set of strings and subsequently arbitrarily choosing strings from within this first set. Another approach would be to memorize the generated strings underway, and then in later calls flip a coin to decide between generating a new one or returning one of the memorized ones. Ideally, `qcheck`'s built-in string generator should use such a strategy to increase the chance of testing code depending on identical strings without limiting itself to a fixed set. In the absence of such a built-in generator we settled for the above approach, which turned out to be good enough for exposing a new class of errors in our analysis.

First, `lookup_prop` in `storelattice` now failed monotonicity with the new generators. On closer inspection, the issue turned out to be a problem of `proplattice`: for the generated counterexample, a key `"b"` would be absent from a smaller table `p.table` (leading to lookup returning the default), whereas the key would be present in the bigger table `p'.table` (leading to returning the corresponding table entry). As a result, the two results were incomparable. This led us to revise our `arb_elem_le` generator, such that any removed entries in the smaller table would be accounted for by the generated default entries. This way, the generated pair would actually be ordered by our intended `proplattice` ordering. Fixing this was not enough to satisfy our quickcheck approach, though. Since join-meet absorption was failing, we first revised the meet operation to only include statically known entries in the result with the `absencelattice` lattice bit being set to present, when such entries were definitely present in either (a) both arguments or (b) in one argument and accounted for by the other's defaults. There were still generated tests failing join-meet absorption, however. This happened when a certain entry from one `proplattice` value was definitely not in the other default's over-approximation. In this situation, we fixed the overall result to be bottom, as no concrete tables exist in the intersection: no concrete table exists that both *definitely has an entry* and *definitely does not have the entry*. This improved understanding of the desired lattice order again led us to revise two aspects of our tests:

- the ordering algorithm behind `proplattice`'s `leq` operation (described above) such that definite, statically known entries of the intended bigger table were also checked to be present in the indended smaller table (otherwise they would not be properly ordered), and
- `arb_elem_le` such that it would not return an intended smaller `proplattice` value with definitely present keys removed from its argument table (again, as otherwise the generated `proplattice` value would not be smaller).

DOI: 10.1002/stvr

Alas, with these fixes in place, the join of `proplattice` was now failing to be associative. Quickchecking produced a counterexample of three `proplattice` values, where joining them in either way would give the same default entries, but not the same statically known table entries. This was because a join in one order pushed a collective `default_key` above a certain entry, so that the default would be included for that particular entry, whereas it would not when the `proplattice` values were joined in the opposite order. Ultimately, we fixed the issue by revising the design of `proplattice` to include an additional `default_str` entry, as explained in Section 2.3:

```
default_str : VL.elem;
```

This new field works as a fall-back for *string entries only* and supplements the above `default_key` and `default` fields for over-approximating the keys and values of *non-string entries*. This revision furthermore had the advantage of completely decoupling the handling of string and non-string keys. With this revised design we were in business to eliminate the remaining errors.

Under the revised design, join-meet absorption was failing again and quickcheck provided a counterexample of two `proplattice` values `p` and `p'` failing $\texttt{p} \sqcup (\texttt{p} \sqcap \texttt{p'}) = \texttt{p}$. Now the problem was with statically known entries from `p.table` with uncertain presence. If such an entry had an empty (bottom) meet with `p'.default_str`, we would get in trouble by omitting the entry entirely from the resulting table: since the later join of the join-meet absorption property would then include `default_str` in the entry's result, it would ultimately be different from `p.table`'s original entry. The fix here was illuminating: we would instead emit an uncertain ('maybe absent'), bottom entry for such entries, thus preventing a fall back to `default_str` and thereby expressing *certain absence* of the entry in an otherwise over-approximating domain!

At this point all lattice modules passed the generic test suite. Two `statelattice` operation tests failed monotonicity in their `statelattice` argument: `read_name` and `write_name` but only in 1–2 out of 1000 generated cases and with very long counterexamples generated. We therefore increased the number of generated tests to increase the chance of generating a small counterexample. The smallest one we found had 174 lines! In the end, these two errors were easy to localize and fix, given our new gained understanding of the lattice ordering. Since `proplattice` values double as environments, their order should be compatible with operations for program variables such as `read_name` and `write_name`. We therefore adjusted the operations to continue lookup for variable entries not available with certainty (an artifact of the chosen double representation).

The nature of errors found This case study answers our first research question in the affirmative. As the above examples also illustrate there is a large span in the nature of the errors we have found using the approach, ranging from simple input-output requirements (like `unop` which could easily be patched to return `bot` when applied to a `bot` argument) to complex ones that required rethinking a domain's entire design (as in our `proplattice` story). The error in the former example could in principle have been caught using a traditional test suite of hand-written unit tests for all operations. That would be repetitive but straightforward. The error in the latter example would be harder to catch with a traditional unit test approach for several reasons: (1) these test and catch *relational errors* that express how the inputs and outputs of multiple calls have to be related to each other (like our initial `flip` example), (2) they corner the bugs by utilizing a combination of properties (monotonicity, join-meet absorption, associativity of join), all of which would need to be hand-written, and (3) they require that the tester comes up with a number of tricky test inputs for each of the above. In principle one can always hand write the same tests that a machine can generate, but we believe that doing so by hand to reach the same level of confidence is unrealistic. For the purposes of testing static analysis implementations we find the combination of the algebraic specification of properties and quickcheck's random generation of input to be a particularly good fit.

The importance of generators and revisiting program specifications Overall, we found that when using quickchecking, the programmer's increase in confidence in the tested code should be proportional to the quality of the involved generators. As an extreme example, for a static analysis one can satisfy the requirements of our approach by letting `arb_elem` constantly return `bot`, by letting `arb_elem_le` constantly return `bot` for any argument, and by letting `equiv_pair` constantly return a

DOI: 10.1002/stvr

Lattice module	Test suite coverage (in %)	QuickCheck coverage (in %)	Combined coverage (in %)
absencelattice	81	100	100
numberlattice	100	100	100
stringlattice	69	97	100
valuelattice	95	98	100
envlattice	90	100	100
proplattice	56	97	97
storelattice	89	94	98
statelattice	71	99	99
analysislattice	81	95	100

Table I. Coverage of analysis code (excluding tool front-end). The second column lists coverage of our original test suite. The third column lists coverage of the QuickCheck-based tests.

pair of `bot` values. Such generators will obviously not give rise to tests that exercise all plausible execution paths through a static analysis and passing them should not get our hopes up. In our case, we had non-trivial generators in place which had already helped find errors. Improving the generators further helped find and fix even more errors. As illustrated by the above revision, we have generally found that quickchecking is a good reason to (re)consider what properties a lattice operation should have. For another example, following TAJS [7], initially the element type of `proplattice` as described in Section 2.3 did not have an explicit `Bot` element. Instead the bottom element representing an empty Lua table would be implemented as an empty internal OCaml table and bottom defaults. As a consequence, the initial implementation of `PL.find` (identical to version described in Section 2.3 but without the initial dispatch on `Bot`) was not strict for a good reason: looking up an entry in the empty table should return (a sound overapproximation of) `nil`, which is clearly not bottom. By adding an explicit `Bot` element and a dispatch in `PL.find` as part of our revision we thereby removed an exception to our specification. In general, the observation of using quickchecking to revisit program specifications agrees with that of the original QuickCheck authors [2].

Coverage To answer our second research question we devised an experiment to measure coverage of the quickchecked analysis code. To measure coverage we instrumented the source code using the 'Bisect' tool.[§] It works as a preprocessor to OCaml code, by statically annotating program points with labels and dynamically tracking the visited program points. Table I reports the percentages of visited program points. We omit coverage of lattice pretty-printing routines as these are irrelevant to the properties being tested. Column 3 shows that quickchecking achieves reasonable coverage (94–100%). `storelattice` has the lowest percentage, mainly because of the intricate semantics of Lua's 'metatables': these require other tables (and functions) to be installed at special string entries of a table, and the corresponding code is thus not as easily exercised by a generator. The alert reader may wonder how the randomization aspect of quickchecking influences these numbers. By default, `qcheck` seeds the pseudo random number generator identically on each run. We therefore conducted the experiment of seeding it differently (using OCaml's `Random.State.make_self_init()`) on four additional runs, running the quickcheck test suite again, and measuring coverage. Across all five runs, 7 out of 9 lattice modules obtained the exact same coverage as reported in column 3. Only the coverage of `valuelattice`, and `storelattice` differed. Of these two modules `valuelattice` had 98–99% coverage, and `storelattice` had 94–95% coverage. As such, a different (pseudo) randomization does not affect the outcome significantly.

By itself, our test suite consisting of 172 hand-written programs obtains slightly worse coverage (56–100%), as shown in column 2. However, if we combine the two approaches, we achieve full coverage in 6 out of 9 lattices and close to full coverage in general (97–100%). Our interpretation of these numbers is that quickchecking is useful to exercise the esoteric paths in lattice code. The

[§] `http://bisect.x9c.fr/`

DOI: 10.1002/stvr

black-box nature of the approach for testing these paths is obviously no silver bullet. Yet, when combined with a standard test suite of programs, the two complement each other well.

The implementation Our implementation of the testing framework consists of a compact 391 line OCaml module, `LCheck`. This module contains a functor with 19 reusable lattice property tests as well as the EDSL code and a number of lattices (the Boolean lattice and its dual, a pair lattice, and a list lattice). Since this module can be reused across many static analyses it is separately available for download.[¶] Applying the EDSL to the Lua type analysis takes an additional 1213 lines of code. Of those lines, the challenging part is the generator code, spanning 228 lines of code (distributed between 41 lines for reusable operations, such as `build_set`, `le_gen`, and `le_entries`, and 187 lines for `arb_elem`, `arb_elem_le`, and `equiv_pair` for each of the 9 lattices modules listed in Table I). With this code we then check the 9 lattice modules along with the two Boolean lattices and 2 instantiations of both the generic pair and list lattices. In addition we test a total of 119 lattices operations, such as `PL.find` and `Str.concat`, spread across the 9 above lattice modules by way of our signature-based EDSLs (utilizing the Bool lattices and the pair and list lattice instantiations in these signatures). In total these sum up to 871 checked properties, distributed between 290 lattice properties and 581 properties of their associated operations. If we include the reusable EDSL module code, this makes for approximately two lines per checked property.[‖]

In comparison, suppose we wish to test the same properties using hand-written unit tests. To test each of these properties by hand for only one choice of input would thus require 871 hand-written unit tests. A reasonable assumption is that each of these tests takes 2 lines of code on average. We would then spend 1742 lines of code on 871 hand-written unit tests, compared to our 1604 lines of code on 871 000 property-based tests (recall each property is tested on 1000 inputs): three orders of magnitude as many tests with a comparable amount of code! Naturally it is more challenging to program, e.g., a generator for `proplattice` as illustrated by our revision than to write a number of unit tests, nevertheless this estimate gives a natural explanation as to how we are able to achieve greater coverage: many more tests are simply being run.

We have not attempted to quickcheck the analysis's tree-walker over arbitrary syntax trees at this point. Doing so, would open up for testing for soundness — another central property of static analyses. Concretely, we receivedtwo bug reports related to soundness of the Lua analysis. One was related to soundly joining lists of return values (in Lua functions can return a list of results); another was related to a corner case concerning Lua's metatables (caused by an ambiguous sentence in the language specification). Both of these slipped through our hand-written test suite and were not found by quickchecking the underlying lattices. In order to thoroughly test a static analysis, we believe that one should both exercise its core components (the underlying lattices) as well as test it for soundness. As such, developing testing techniques for soundness is both a promising and important path for future work.

We would like to extend the infix syntax to handle multiple properties in *one* test signature. Overall this should lower the lines-of-code/property ratio. Technically, this would require changing the underlying CPS building into one returning a list answer type. We would also like to handle more signature properties, e.g., for extensive functions. In addition, one could consider to revise the current infix syntax using the camlp4 preprocessor, to avoid having to write **`module`** repeatedly and to allow arrow syntax that coincides with OCaml's built-in type signature syntax. For now, we have chosen to keep within the bounds of pure OCaml, to limit the number of dependencies.

We stress that although the current development has taken place within OCaml, it could just as well have been formulated, e.g., with Haskell's type classes. The type-safe embedding of the EDSL utilizes the Hindley-Milner based type system of OCaml, thereby statically preventing type errors in the tests, e.g., after lattice or transfer function revisions. A less type-safe embedding could be implemented, e.g., in Java. One popular application of quickchecking is to test programs written

[¶] `https://github.com/jmid/lcheck`

[‖] The coverage reports, the source code of the analysis, and the tests are available at `https://github.com/jmid/luata-quickcheck`.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

in the dynamically typed programming language Erlang. Indeed, nothing of the present framework mandates a statically typed programming language: the extension of lattice interfaces and an EDSL of signatures could just as well be written in JavaScript, Lua, or Scheme.

6. RELATED WORK

Previous work on increasing confidence in static analyses, range from basic testing to rigorous pen-and-paper proofs such as Astrée's [17]. Our approach can be beneficial to analyses from both ends of the spectrum, to help ensure that an implementation captures the intended meaning — be it in a programmer's mind or in a rigorous pen-and-paper formalization. The growing interest in proof assistants, such as, Coq, led Pichardie et al. [18, 19] to formalize abstract interpretation in constructive logic. Combined with Coq's ability to extract, e.g., OCaml code from its constructive proofs, this minimizes the 'trusted computing base' to Coq itself. In a recent endeavor, Blazy et al. [20] investigated the formalization of a value analysis for C integrated into the CompCert framework. To keep things manageable, this approach relies on a common abstract domain interface akin to a bare bones version of Fig. 2, and it formalizes (and extracts) a *fixed-point verifier* in Coq rather than the fixed-point computing value analysis itself. A rather different approach was taken by Murawski and Yi [21], by developing a static monotonicity analysis formulated as a type-and-effects system. Their analysis conservatively accepts (or rejects) λ-definable functions over finite lattices fed to a static analysis generator. We believe that quickchecking as demonstrated in this paper offers a lightweight alternative to the above approaches. It is type safe, it is reusable, and it supplements basic testing well.

The OCaml implementation of the Lua type analysis benefits from lack of side effects, e.g., assignments. This makes it easier to check and gain confidence in the individual lattice operations as their output is determined solely by the input parameters. The static analysis community has previously benefited from analysis implementations in functional languages: The Astrée static analyzer [17] is implemented in OCaml, the CIL infrastructure for analysis and transformation of C program [22] is implemented in OCaml, Frama-C [23], an industrial-strength static analysis framework for the C programming language, is implemented in OCaml, MathWorks's (formerly, PolySpace Technologies) PolySpace verifier [24] is written in Standard ML, Simon's value analysis of C [25] is implemented in Haskell, and SLAM (subsequently, Static Driver Verifier) was originally implemented in OCaml [26]. There are therefore plenty of existing analysis implementations that might benefit from our methodology. In addition, both Cousot et al. [17] and Jensen et al. [7] report to have arrived at their lattice and transfer function designs after multiple revisions: a potentially fruitful scene for our suggested methodology.

In a follow-up paper to the original, Claessen and Hughes [4] develop a QuickCheck framework for testing monadic code using multiple approaches, including an *algebraic specification* and a *model-based specification* approach. They formulate a little EDSL to express (and generate) arbitrary context traces of an abstract data type (ADT) under test, which they then use for both specification approaches. One of their key insights of their algebraic specification approach is to characterize observational equivalence of such code ("in all contexts, one piece of imperative code is indistinguishable from another"), in terms of the trace EDSL (for arbitrarily generated context traces, perform an equality test) which one can easily test for. Their second model-based specification approach ensures agreement between the result of each ADT operation and the corresponding operation over a simple model, when run in an arbitrary trace context (again expressed with the trace EDSL). Their model-based approach has since become so common to test both stateless and stateful code that the commercial Erlang QuickCheck library supports a *state machine framework* for easily expressing an ADT, a model, and the agreement between transitions over both [27]. As our analysis computes over approximate states, it has an imperative flavor similar to the examples of Claessen and Hughes [4]. For example, we would like to investigate methods for generating relational lattice values (e.g., `valuelattice` elements whose sets of labels all belong to the generated `storelattice` value). Furthermore, the tree-walker of the Lua type analysis is written monadically, and hence should be a likely target for their techniques. Potentially this could utilize some of the

Prepared using stvrauth.cls
DOI: 10.1002/stvr

techniques of Pałka et al. [28] for generating random abstract syntax trees. Rather than taking our current *white-box* testing approach, which utilizes full knowledge of the inner workings of each lattice module, it would also be interesting to investigate a *black-box* testing approach akin to Claessen and Hughes's ADT context traces, by generating a symbolic sequence (or more accurately a tree, since we include binary operations) of operations (`join`, `meet`, etc.) and testing the algebraic lattice laws for arbitrary lattice values arising from these. This would have the advantage of letting each module preserve module-local invariants without random generation potentially interfering with these. On the other hand, each lattice has specific operations (a function lattice has lookup, an interval lattice has abstract arithmetic, etc.), which would break the uniform design behind the current approach.

As discussed in Section 5, some errors found in our case study are relational, in the sense that they involve the relation between the inputs and outputs of multiple calls. Metamorphic testing is a variant of property-based testing that focuses on such relational properties [29].

Independently from the seminal work by Claessen and Hughes [4] on the model-based specification approach in the functional programming community, Offutt and Abdurazik [30] suggested *model-based testing* to the software engineering community: an approach of letting test generation be driven by (UML) statecharts. Although the two were thus presented in widely different clothes—Haskell and UML—at their core the underlying idea was the same. The initial presentations differed in that Offutt and Abdurazik would let their generator accept coverage information in a feedback loop, and in that the generated tests would be stored offline, to have redundant cases removed, and only be run later. Finally Offutt and Abdurazik's approach is close to the later state machine framework of the commercial Erlang QuickCheck library [27] in that state charts are not only used to drive generation: in both cases state transitions are annotated with predicates that capture the properties to test for. For imperative lattice implementations it could be relevant to let their generation be driven by such state transition models.

In the context of testing abstract datatypes, Holdermans [15] recalls how naively lifting axioms from an algebraic specification to quickcheck properties leaves programmers with a false sense of security: a buggy implementation can still pass a seemingly complete property-based test suite. In the pitfall he investigates, the randomly generated tests are insufficient to cover all concrete representations of an abstract data type. As a remedy he extends the test suite with invariance tests. We have chosen to follow Holdermans's recommendation in our work.

In the broader programming language community, a number of tools utilize randomized testing. In the field of compiler testing, we briefly touched upon the work of Pałka et al. [28] used to test the Glasgow Haskell Compiler's strictness analyser. Another prominent representative is Csmith by Yang et al. [31]: a tool that uses randomized differential testing to generate random C programs free of C's notorious undefined behavior, yet capable of finding numerous errors in production compilers. Cuoq et al. [32] utilize Csmith specifically for testing the Frama-C static analysis tool using a range of approaches: A first approach modifies Frama-C into an interpreter without abstraction that can be tested against an ordinary C compiler. A second approach emits an inferred analysis invariant as a C assertion for each program and tests whether the assertion holds when executing the decorated program. A third approach performs a constant propagation analysis and similarly tests whether the execution of each program agrees with and without expressions being replaced by constant values discovered by the analysis. PLT Redex [33] is a general tool for semantics engineering that can also quickcheck properties on randomly generated input. In contrast to OCaml's statically typed programs and properties, PLT Redex's internal language is dynamically typed. In the presence of (almost) full coverage, one can argue that static typing plays a less dominant role. Static types, however, provide pedagogical guidelines for composing composite lattice generators out of simpler ones. St-Amour and Toronto [34] report on quickchecking the base type environment of Typed Racket using PLT Redex. Specifically, they check whether the type signature of arithmetic primitives soundly account for their corresponding runtime behavior when applied to randomly generated terms. Overall they report that quickchecking supplements manual testing well and requires low effort, which aligns with our experience.

DOI: 10.1002/stvr

Hriţcu et al. [35] quickcheck an information flow architecture in the form of a simple abstract machine to find counterexamples of non-interference. In doing so, they experiment with (a) different strategies for generation—from naive generators to more clever ones generating pairs of related programs (instruction sequences), and (b) different non-interference properties—from full end-to-end machine execution to single-step machine execution. Our generation of ordered and equivalent lattice element pairs is reminiscent of Hriţcu et al.'s generators of related programs. Our work differs in that it concerns testing general static analysis properties, whereas the work by Hriţcu et al. concerns testing soundness properties of a dynamic analysis.

Numerous techniques have been suggested for automated testing in related settings. For example, Randoop [36] performs feedback-directed random testing for object-oriented programs, and Korat [37] generates test inputs based on formal specifications of pre- and post-conditions. Such techniques can in principle also be applied to test static analyses, however, unlike our approach, they do not exploit the underlying structured domains and the algebraic properties common in static analysis.

The analysis implementation grew out of our earlier study of definitional interpreters for the Lua programming language [38]. The present type analysis provides a sound approximation of the semantics of those interpreters, albeit it supports a substantially bigger language. The Lua type analysis is heavily inspired by TAJS [7], a type analysis for JavaScript. In contrast to the present functional OCaml implementation, TAJS is implemented in Java. Despite extensive testing, the manually developed test suite for TAJS does not achieve full coverage of, for example, its value lattice domain, which plays a central role. This is partially due to algorithmic interference: TAJS's worklist algorithm decides when states should be joined in the lattice, which makes it hard to craft an input program that will force the abstract interpreter down a certain lattice code path. Although unit tests have also been made specifically for that part of the code, not enough resources have been invested in making them sufficiently comprehensive. However, quickchecking is well suited to test such paths extensively and with much less effort.

7. CONCLUSION

We have presented a lightweight methodology for quickchecking static analyses to check a range of properties, and as a result raise confidence in their implementation. We can do so in a non-intrusive and scalable manner: lattice properties of both basic lattices and complex compositional lattices share the same property tests and can quickly be checked on thousands of generated inputs. To quickcheck lattice operations we have developed a type-safe EDSL for expressing common properties. With our EDSL, much of the infrastructure becomes reusable across analyses, and testing a lattice operation property involves little more than writing a type signature.

Based on this positive experience, we encourage static analysis developers to quickcheck their next analysis tool for many of the generic properties that may otherwise be cumbersome to test. Our OCaml-based EDSL may serve as a useful foundation, or as a source for inspiration if using other languages.

ACKNOWLEDGEMENTS

We thank Magnus Madsen and the anonymous ICST and STVR reviewers for comments. In particular, we are grateful for remarks from STVR reviewer 3 that led us to revise our generators and ultimately find even more errors. The work also benefited from Simon Cruanes's `qcheck` quickchecking library and from Xavier Clerc's `Bisect` coverage analysis tool.

REFERENCES

1. Nielson F, Nielson HR, Hankin C. *Principles of Program Analysis*. Springer-Verlag, 1999.
2. Claessen K, Hughes J. QuickCheck: A lightweight tool for random testing of Haskell programs. *Proc. of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, Wadler P (ed.), 2000; 53–64.
3. Midtgaard J, Møller A. Quickchecking static analysis properties. *8th IEEE International Conference on Software Testing, Verification and Validation, ICST'15*, Fraser G, Marinov D (eds.), IEEE Computer Society, 2015.
4. Claessen K, Hughes J. Testing monadic code with QuickCheck. *SIGPLAN Notices* 2002; **37**(12):47–59.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

5. Hughes J. Experiences with QuickCheck: Testing the hard stuff and staying sane. *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, *LNCS*, vol. 9600, Lindley S, McBride C, Trinder PW, Sannella D (eds.), Springer-Verlag, 2016; 169–186.
6. Ierusalimschy R, de Figueiredo LH, Celes W. The evolution of Lua. *Proc. of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, 2007; 2–1–2–26.
7. Jensen SH, Møller A, Thiemann P. Type analysis for JavaScript. *Static Analysis, 16th International Symposium, SAS 2009*, *LNCS*, vol. 5673, Palsberg J, Su Z (eds.), Springer-Verlag, 2009; 238–255.
8. Jones ND, Muchnick SS. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. *Proc. of the Ninth Annual ACM Symposium on Principles of Programming Languages*, DeMillo R (ed.), 1982; 66–74.
9. Chase DR, Wegman M, Zadeck KF. Analysis of pointers and structures. *Proc. of the ACM SIGPLAN 1990 Conference on Programming Languages Design and Implementation*, Fischer BN (ed.), 1990; 296–310.
10. Kildall G. A unified approach to global program optimization. *Proc. of the First Annual ACM Symposium on Principles of Programming Languages*, Ullman JD (ed.), 1973; 194–206.
11. Cousot P, Cousot R. Static determination of dynamic properties of programs. *Proc. of the Second International Symposium on Programming*, 1976; 106–130.
12. Cousot P, Cousot R. Abstract interpretation and application to logic programs. *Journal of Logic Programming* 1992; **13**(2–3):103–179.
13. Adams S. Efficient sets—a balancing act. *Journal of Functional Programming* 1993; **3**(4):553–561.
14. Granger P. Static analyses of congruence properties on rational numbers (extended abstract). *Static Analysis, 4th International Symposium, SAS '97*, *LNCS*, vol. 1302, Hentenryck PV (ed.), Springer-Verlag, 1997; 278–292.
15. Holdermans S. Random testing of purely functional datatypes. Schrijvers and Peña [39]; 275–284.
16. Danvy O. Functional unparsing. *Journal of Functional Programming* 1998; **8**(6):621–625.
17. Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X. The ASTRÉE Analyser. *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005*, *LNCS*, vol. 2618, Sagiv M (ed.), Springer-Verlag, 2005; 21–30.
18. Cachera D, Jensen T, Pichardie D, Rusu V. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science* 2005; **342**(1):56–78.
19. Pichardie D. Interprétation abstraite en logique intuitioniste: extraction d'analyseurs Java certifiés. PhD Thesis, Université de Rennes 1, Sep 2005.
20. Blazy S, Laporte V, Maroneze A, Pichardie D. Formal verification of a C value analysis based on abstract interpretation. *Static Analysis, 20th International Symposium, SAS 2013*, *LNCS*, vol. 7935, Logozzo F, Fähndrich M (eds.), Springer-Verlag, 2013; 324–344.
21. Murawski AS, Yi K. Static monotonicity analysis for λ-definable functions over lattices. *VMCAI*, *LNCS*, vol. 2294, Cortesi A (ed.), Springer-Verlag, 2002; 139–153.
22. Necula GC, McPeak S, Rahul SP, Weimer W. CIL: Intermediate language and tools for analysis and transformation of C programs. *CC'02: Proc. of the 11th International Conference on Compiler Construction*, *LNCS*, vol. 2304, Horspool RN (ed.), Springer-Verlag, 2002; 213–228.
23. Cuoq P, Signoles J, Baudin P, Bonichon R, Canet G, Correnson L, Monate B, Prevosto V, Puccetti A. Experience report: OCaml for an industrial-strength static analysis framework. *Proc. of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP'09)*, Hutton G, Tolmach AP (eds.), 2009; 281–286.
24. Deutsch A. Static verification of dynamic properties, 2003. PolySpace Technologies, white paper.
25. Simon A. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. Springer, 2008.
26. Ball T, Rajamani SK. The SLAM project: Debugging system software via static analysis. *Proc. of the 29th Annual ACM Symposium on Principles of Programming Languages*, Mitchell JC (ed.), 2002; 1–3.
27. Hughes J. Software testing with QuickCheck. *Central European Functional Programming School - Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, *LNCS*, vol. 6299, Horváth Z, Plasmeijer R, Zsók V (eds.), Springer-Verlag, 2010; 183–223.
28. Pałka MH, Claessen K, Russo A, Hughes J. Testing an optimising compiler by generating random lambda terms. *Proc. of the 6th International Workshop on Automation of Software Test, AST 2011*, 2011; 91–97.
29. Cadar C, Donaldson AF. Analysing the program analyser. *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*, Dillon LK, Visser W, Williams L (eds.), ACM, 2016; 765–768, doi:10.1145/2889160.2889206. URL `http://doi.acm.org/10.1145/2889160.2889206`.
30. Offutt AJ, Abdurazik A. Generating tests from UML specifications. *«UML»'99: The Unified Modeling Language - Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, Proceedings*, *LNCS*, vol. 1723, France RB, Rumpe B (eds.), Springer-Verlag, 1999; 416–429.
31. Yang X, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. *Proc. of the ACM SIGPLAN 2011 Conference on Programming Languages Design and Implementation*, Padua D (ed.), 2011; 283–294.
32. Cuoq P, Monate B, Pacalet A, Prevosto V, Regehr J, Yakobowski B, Yang X. Testing static analyzers with randomly generated programs. *NASA Formal Methods - 4th International Symposium, NFM 2012. Proc.*, *LNCS*, vol. 7226, Springer-Verlag, 2012; 120–125.
33. Felleisen M, Findler RB, Flatt M. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.
34. St-Amour V, Toronto N. Experience report: Applying random testing to a base type environment. Morrisett and Uustalu [40]; 351–356.
35. Hriţcu C, Hughes J, Pierce BC, Spector-Zabusky A, Vytiniotis D, Azevedo de Amorim A, Lampropoulos L. Testing noninterference, quickly. Morrisett and Uustalu [40]; 455–468.
36. Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-directed random test generation. *29th International Conference on Software Engineering (ICSE)*, 2007; 75–84.

DOI: 10.1002/stvr

37. Boyapati C, Khurshid S, Marinov D. Korat: automated testing based on Java predicates. *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, 2002; 123–133.
38. Midtgaard J, Ramsey N, Larsen B. Engineering definitional interpreters. Schrijvers and Peña [39]; 121–132.
39. Schrijvers T, Peña R ((eds.)). *PPDP'13: Proc. of the 15th International Symposium on Principles and Practice of Declarative Programming*, 2013.
40. Morrisett G, Uustalu T ((eds.)). *Proc. of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, 2013.

A. APPENDIX

A.1. A simplified grammar for Lua

$$
\begin{aligned}
\mathit{lit} ::= {} & \texttt{nil} \\
\mid {} & \mathit{bool} \\
\mid {} & \mathit{string} \\
\mid {} & \mathit{number} \\
\mid {} & \texttt{\{}\,\mathit{exp}^{*}\,\texttt{;}\ (\mathit{id}\texttt{=}\mathit{exp})^{*}\,\texttt{\}} \\
\mid {} & \texttt{function (}\mathit{id}^{*}\texttt{)}\ \mathit{block}\ \texttt{end}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{exp} ::= {} & \mathit{lit} \\
\mid {} & \mathit{lvalue} \\
\mid {} & \mathit{unop}\ \mathit{exp} \\
\mid {} & \mathit{exp}\ \mathit{binop}\ \mathit{exp} \\
\mid {} & \mathit{exp}\ \texttt{and}\ \mathit{exp} \\
\mid {} & \mathit{exp}\ \texttt{or}\ \mathit{exp} \\
\mid {} & \mathit{exp}\,\texttt{(}\mathit{exp}^{*}\texttt{)} \\
\mid {} & \mathit{exp}\texttt{.}\mathit{id}\,\texttt{(}\mathit{exp}^{*}\texttt{)} \\
\mid {} & \texttt{(}\mathit{exp}\texttt{)}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{lvalue} ::= {} & \mathit{id} \\
\mid {} & \mathit{exp}\texttt{.}\mathit{id} \\
\mid {} & \mathit{exp}\,\texttt{[}\mathit{exp}\texttt{]}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{stmt} ::= {} & \texttt{break} \\
\mid {} & \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{block}\ \texttt{else}\ \mathit{block}\ \texttt{end} \\
\mid {} & \texttt{while}\ \mathit{exp}\ \texttt{do}\ \mathit{block}\ \texttt{end} \\
\mid {} & \texttt{do}\ \mathit{block}\ \texttt{end} \\
\mid {} & \mathit{lvalue}^{+} = \mathit{exp}^{+} \\
\mid {} & \texttt{local}\ \mathit{exp}^{+}\ (= \mathit{exp}^{+})? \\
\mid {} & \mathit{exp}\,\texttt{(}\mathit{exp}^{*}\texttt{)} \\
\mid {} & \mathit{exp}\texttt{.}\mathit{id}\,\texttt{(}\mathit{exp}^{*}\texttt{)} \\
\mid {} & \texttt{return}\ \mathit{exp}^{*}
\end{aligned}
$$

$$
\mathit{block} ::= \mathit{stmt}^{*}
$$

A simplified BNF grammar of Lua. ?, $*$, and $+$ denote optional elements, empty and nonempty Kleene sequences, respectively.

A.2. EDSL signatures

Below we include selected signatures of the EDSL.

```
(*  The reusable functor of generic lattice tests (top not required) *)
module GenericTests :
  functor (L : LATTICE_TOPLESS) ->
    sig
      val arb_pair : (L.elem * L.elem) QCheck.Arbitrary.t
      val arb_triple : (L.elem * L.elem * L.elem) QCheck.Arbitrary.t
      val ord_triple : (L.elem * L.elem * L.elem) QCheck.Arbitrary.t
      val pp_triple : (L.elem * L.elem * L.elem) QCheck.PP.t
      val size : L.elem -> int
      val size_pair : L.elem * L.elem -> int
      val size_triple : L.elem * L.elem * L.elem -> int
      val leq_refl : QCheck.test
      val leq_trans : QCheck.test
      val leq_antisym : QCheck.test
      val bot_is_lowerbound : QCheck.test
      val join_comm : QCheck.test
      val join_assoc : QCheck.test
      val join_idempotent : QCheck.test
      val meet_comm : QCheck.test
      val meet_assoc : QCheck.test
      val meet_idempotent : QCheck.test
      val join_meet_absorption : QCheck.test
      val meet_join_absorption : QCheck.test
      val leq_compat_join : QCheck.test
      val join_compat_leq : QCheck.test
```


DOI: 10.1002/stvr

```
      val join_compat_meet : QCheck.test
      val meet_compat_join : QCheck.test
      val meet_compat_leq : QCheck.test
      val leq_compat_meet : QCheck.test
      val check_ordering : QCheck.test
      val pp_pair : (L.elem * L.elem) QCheck.PP.t
      val ord_pair : (L.elem * L.elem) QCheck.Arbitrary.t
      val suite : QCheck.test list
    end

(*  The reusable functor of generic lattice tests (top required)  *)
module GenericTopTests :
  functor (L : LATTICE) ->
    sig
      val top_is_upperbound : QCheck.test
      val suite : QCheck.test list
    end


(*  The combinator for expressing a monotonicity test  *)
val op_monotone : (module LATTICE_TOPLESS with type elem = 'a) ->
                  (module LATTICE_TOPLESS with type elem = 'b) ->
                  (('a * 'a) QCheck.PP.t * ('a * 'a) QCheck.Arbitrary.t *
                   (('a -> 'b) -> ('a * 'a) QCheck.Prop.t) * string * int -> 'c) -> 'c

(*  The combinator for expressing an invariance test  *)
val op_invariant : (module LATTICE_TOPLESS with type elem = 'a) ->
                   (module LATTICE_TOPLESS with type elem = 'b) ->
                   (('a * 'a) QCheck.PP.t * ('a * 'a) QCheck.Arbitrary.t *
                    (('a -> 'b) -> ('a * 'a) QCheck.Prop.t) * string * int -> 'c) -> 'c

(*  The combinator for expressing a strictness test  *)
val op_strict : (module LATTICE_TOPLESS with type elem = 'a) ->
                (module LATTICE_TOPLESS with type elem = 'b) ->
                ('a QCheck.PP.t * 'a QCheck.Arbitrary.t *
                 (('a -> 'b) -> 'a QCheck.Prop.t) * string * int -> 'c) -> 'c

(*  The combinator for expressing a distributivity test  *)
val op_distributive : (module LCheck.LATTICE_TOPLESS with type elem = 'a) ->
                      (module LCheck.LATTICE_TOPLESS with type elem = 'b) ->
                      (('a * 'a) QCheck.PP.t * ('a * 'a) QCheck.Arbitrary.t *
                       (('a -> 'b) -> ('a * 'a) QCheck.Prop.t) * string * int -> 'c) -> 'c

(*  The combinator for adding additional arguments to the left  *)
val pw_left : (module ARB_ARG with type elem = 'a) ->
              ((module LATTICE_TOPLESS with type elem = 'b) ->
               (module LATTICE_TOPLESS with type elem = 'c) ->
               ('d QCheck.PP.t * 'd QCheck.Arbitrary.t *
                (('f -> 'g) -> 'd QCheck.Prop.t) * string * int -> 'e) -> 'h) ->
              (module LATTICE_TOPLESS with type elem = 'b) ->
              (module LATTICE_TOPLESS with type elem = 'c) ->
              (('a * 'd) QCheck.PP.t * ('a * 'd) QCheck.Arbitrary.t *
               (('a -> 'f -> 'g) -> ('a * 'd) QCheck.Prop.t) * string * int -> 'e) -> 'h

(*  The combinator for adding additional arguments to the right  *)
val pw_right : (module ARB_ARG with type elem = 'a) ->
               ((module LATTICE_TOPLESS with type elem = 'b) ->
                (module LATTICE_TOPLESS with type elem = 'c) ->
                ('d QCheck.PP.t * 'd QCheck.Arbitrary.t *
                 (('f -> 'g) -> 'd QCheck.Prop.t) * string * int -> 'e) -> 'h) ->
               (module LATTICE_TOPLESS with type elem = 'b) ->
               (module LATTICE_TOPLESS with type elem = 'c) ->
               (('d * 'a) QCheck.PP.t * ('d * 'a) QCheck.Arbitrary.t *
                (('f -> 'a -> 'g) -> ('d * 'a) QCheck.Prop.t) * string * int -> 'e) -> 'h

(*  The combinator ceiling off a lattice signature test  *)
val finalize : (('a QCheck.PP.t * 'a QCheck.Arbitrary.t *
                 (('b -> 'c) -> 'a QCheck.Prop.t) * string * int -> QCheck.test) ->
                QCheck.test) -> string * ('b -> 'c) -> QCheck.test

(*  Infix syntax for finalize  *)
```


DOI: 10.1002/stvr

```
val ( =:: ) : (('a QCheck.PP.t * 'a QCheck.Arbitrary.t *
                (('b -> 'c) -> 'a QCheck.Prop.t) * string * int -> QCheck.test) ->
               QCheck.test) -> string * ('b -> 'c) -> QCheck.test

(*  A reusable Boolean lattice ordered under reverse implication ordering  *)
module Bool  : LATTICE

(*  A reusable Boolean lattice ordered under implication ordering  *)
module DBool : LATTICE

(*  A reusable functor for creating pair lattices  *)
module MkPairLattice : functor (A : LATTICE_TOPLESS with type elem = 'a)
                               (B : LATTICE_TOPLESS with type elem = 'b) ->
                               LATTICE_TOPLESS with type elem = 'a * 'b

(*  A reusable functor for creating list lattices  *)
module MkListLattice : functor (A : LATTICE_TOPLESS with type elem = 'a) ->
                               LATTICE_TOPLESS with type elem = 'a list
```


DOI: 10.1002/stvr